Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра математики

Кваліфікаційна робота

освітній ступінь – бакалавр

на тему: **«Математичне моделювання та створення спеціалізованих програмних засобів для перевірки C++ та Java програм на плагіат»**

Виконав: студент 4-го року навчання
освітньої програми «Прикладна математика»,
спеціальності 113 Прикладна математика

Ляшко Андрій Володимирович

Керівник: Жежерун О. П.,
доцент, к.н.

Рецензент_____
(прізвище та ініціали)

Кваліфікаційна робота захищена
з оцінкою _____

Секретар ЕК _____
«____» _____ 20____ р.

Київ – 2021

Національний університет «Києво-Могилянська академія»

Факультет інформатики
Кафедра математики
Освітній ступінь бакалавр
Спеціальність 113 Прикладна математика
(шифр і назва)

ЗАТВЕРДЖУЮ
Т.в.о. завідувача кафедри
доцент, к.н. Р. К. Чорней

“____” _____20___року

З А В Д А Н Н Я

ДЛЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ СТУДЕНТУ

Ляшко Андрію Володимировичу
(прізвище, ім’я, по батькові)

Тема роботи: **Математичне моделювання та створення спеціалізованих програмних засобів для перевірки C++ та Java програм на плагіат**

керівник роботи: Жежерун Олександр Петрович, доцент, кандидат фіз.-мат. наук
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від « 16 » жовтня 2020 року № 890-с
2. Строк подання студентом роботи 28 травня 2021 року
3. План роботи:
Анотація

Вступ

Способи транспонування коду на мовах C++ та Java

Вступ у лямбда числення

Переведення програм у лямбда терми

Порівняння програм, переведених у лямбда числення

Висновок

Список використаної літератури

Зміст **1**

Зміст:

Анотація

Ця дипломна робота присвячена темі виявлення плагіату в студентських, а за можливості і в іншому, ширшому середовищі, робіт, які написані на таких мовах програмування як C++ та Java. В подальшому до переліку можна додати й інші мови програмування, такі як Python та інші.

Перший розділ буде присвячено аналізу схожості та відмінності між кодом, написаним на двох ключових мовах. В другому розділі ми поговоримо про ламбда терми, яку роль вони грають в програмування та буде пояснення, чому темою дипломної роботи була обрана ця тема, та який зв’язок між математичним моделюванням та нашою роботою.

У другому розділі ми переглянемо найбільш вживані та прості методи транспонування програм, розберемо що таке лямбда числення, як представити програму у вигляді лямбда терму, що таке рівність та еквівалентність лямбда термів, які є методи переходу до еквівалентного терму. Також ми спробуємо виявити транспонування коду двома способами.

Вступ

Тож зосередимось на більш поглибленних методах плагіату таких як використання посилань, указників та інша робота з пам’яттю та інші фундаментальні відмінності в обраних мовах.

Уявимо собі ситуацію, коли є певне завдання, яке потрібно виконати на одній з цих двох мов. Завжди знайдуться ті, хто захоче піти легким шляхом та спробувати знайти вже готову програму та схитрувати, а не слідувати складному шляху та розбиратись в теорії і намагатись реалізувати її на практиці.

Більшість програм можна знайти готовими, в різних їх різновидах, налаштуваннях і тд… Проте коли потрібно запрограмувати наприклад якусь математичну функцію, яку майже ніхто не використовує, виникає проблема, як та де знайти інформацію. Після довгих пошуків було знайдено потрібний код, але на іншій мові програмування. Будемо вважати, що ця людина має достатньо навичок, щоб перенести код з однієї мови на іншу. Тож задача нечесного студента переписати код так, щоб ніхто не здогадався про ідентичність коду, наша ж задача полягає в виявленні потенційно схожого коду та привернення уваги викладача до проблемних місць. Жодна програма не дає сто відсоткової ймовірності виявити схожий код, звичайно ж якщо він не є абсолютно ідентичним. З одного боку код може бути схожим, з іншого боку, коли багато студентів робить однакове завдання, то між собою код повинен бути хоч трохи подібним. Тож ми повинні зібрати інформацію та передати її викладачу, який потім сам буде вирішувати що з нею робити.

Може здатися що можна просто всі програми помічати схожими та надавати цю інформацію, проте тоді втрачається сенс потрібності програми. Саме такі випадки в загальному ми найбільше будемо розглядати в першому розділі.

У другому розділі ми поговоримо про те, що таке лямбда терми, як можна використати їх для виявлення схожості програм, чи будь-яку програму можна представити як лямбда терм та який принцип порівняння схожості двох лямбда термів.

Розділ 1: схожість та відмінність програм, написаних на мові C++ та Java

В першому розділі ми зосередимось на відмінностях в цих мовах програмування. Найголовнішою відмінністю є те, що в Java не потрібно думати про роботу з пам'яттю в той час як в C++ ця можливість надана програмісту, який сам має контролювати ці процеси.

1.1 Посилання, указники та робота з пам'яттю

Отже Java не підтримує можливість роботи з пам'яттю. ЇЇ використовують коли швидкість роботи не грає великої ролі. Це робить її більш легкою для вивчення в якості першої мови програмування. Яскравим прикладом використання Java є андроід розробка. З іншої сторони, C++ є більш швидкою. Його потенціал розкривається при створенні операційних систем, драйверів пристроїв та іншого програмного забезпечення де швидкість важлива.

Отже в цій темі ми зосередимось на C++. Вона є більш складною мовою для освоєння, через більшу кількість роботи можна зробити більшу кількість помилок, в тому числі неправильно працювати з пам'яттю(програмісти-початківці часто помиляються при роботі з пам'яттю, наприклад змінюють об'єкт за неправильною адресою, що в свою чергу викликає ланцюгову реакцію і неправильні результати роботи програми).

Тож, повертаючись до нашої теми, для плагіату є дуже велика кількість методів, які не змінять відповідь роботи програми.

Не будемо розбирати що таке посилання, що таке указник, скажемо лише, в чому їх принципова різниця. Перша відмінність це те, що указник не обов'язково ініціалізувати якимось значенням на відміну від посилання.

Друга, це те, що указник може змінюватись за необхідності необмежену кількість разів в той час як посилання після прив’язки до адреси пам’яті не може бути прив’язана до іншої комірки.

Тепер розглянемо приклад. У нас є проста програма, яка зчитує значення двох змінних та присвоює результат додавання в ще одну змінну.

```
int main() {
     int a = 0, b = 0;
     cin >> a >> b;
     int c = a + b;
     cout << c << endl;
     return 0;
}
```

Тепер потрібно переробити її якимось чином. Можна скористатись методами, які є універсальними, та їх можна використати в багатьох мовах, такі як перейменувати змінні чи змінити тип так, щоб результат був той самий. Ми не будемо розглядати такі випадки, а одразу перейдемо до тих, які можна використати саме в C++ або Java. Отже, можна просто створити посилання на вже обрахований результат та при виведенні його розіменувати.

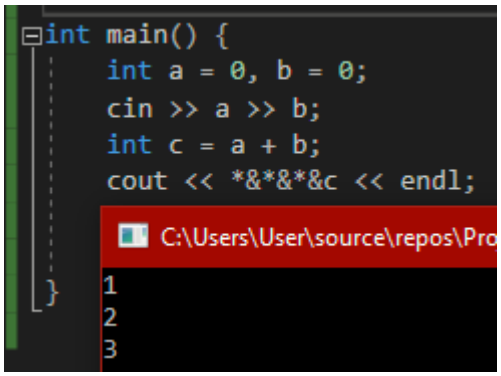
```
int main() {
     int a = 0, b = 0;
     cin >> a >> b;
     int c = a + b;
     int *d = &c;
     cout << *d << endl;
     return 0;

    }
```

Тепер ми будемо мати посилання d на об’єкт c, точніше на його значення. Виводом буде не значення об’єкта c напряму, а роз’іменоване посилання на нього, результат роботи програми не зміниться, проте код був змінений. Операцію роз’іменування посилань можна використовувати безліч разів, тільки на кожне посилання потрібно своя операція роз’іменування.

Подивимось наступний приклад.

```
int main() {
     int a = 0, b = 0;
     cin >> a >> b;
     int c = a + b;
     cout << *&*&*&c << endl;
     return 0;
}
```

З першого погляду здається що це неправильно, та й хто буде так робити, проте це є правильним. Якщо спробувати запустити код та отримати результат, будемо мати:



Людина одразу побачить що код переписаний та дороблений, проте програма може не виявити різницю. Також програму можна переписати з допомогою посилань указників та додаткових змінних так, що це не буде так помітно, але суть змін буде однакова, тож нам потрібно розробити інструмент, який буде вміти це виявляти.

1.2 Перевантаження операторів

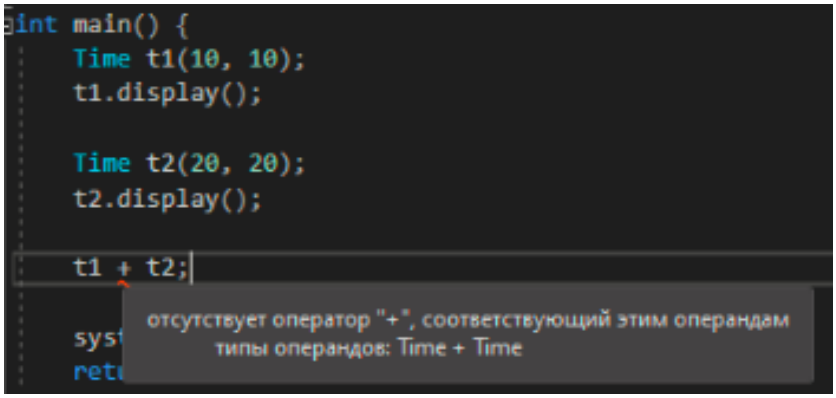
Однією важливою функцією, яка працює в C++, проте немає в Java – це перевантаження операторів. Її використовують для перевизначення дії, яка відбувається між двома об'єктами.

Перевантажити оператори можна лише ті, які вже визначені в C++. Створити власний оператор не дозволено. Оператор можна визначити як в середині класу, так і ззовні. Ця функція має велику користь, коли потрібно визначити певну дію для двох об'єктів, коли, наприклад, потрібна операція додавання двох об'єктів. Програма не знає що таке об'єкт, якого він типу, а отже й не знає за яким принципом його додавати.

Приведемо простий приклад використання перевизначення оператора додавання та оператора виведення. Припустимо що в нас є клас Time, який зберігає час в хвилинах та секундах. Для захисту параметри що передаються повинні бути константними та невід'ємними. Створимо два об'єкта типу Time, і виведемо результат їх додавання.

```
class Time {
private:
     int _hours = 0;
     int _minutes = 0;
public:
     Time(unsigned const int hours, unsigned const int
minutes):_hours(hours+minutes/60), _minutes(minutes%60){}
     void display() {
          cout << "hours = " << this->_hours << endl <<
"minutes = " << this->_minutes << endl;
     }
};
int main() {
     Time t1(10, 10);
     t1.display();
     Time t2(20, 20);
     t2.display();
     return 0;
    }
```

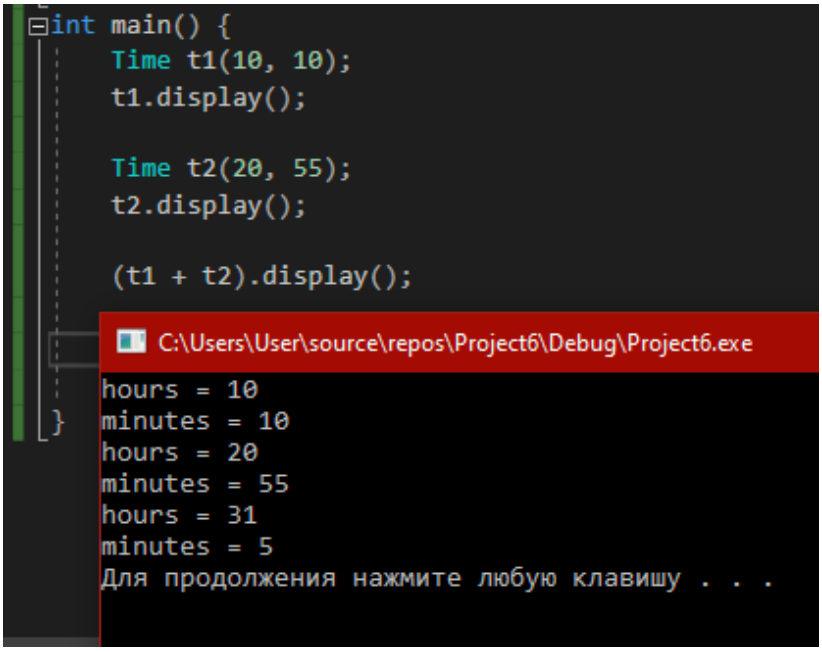
Так виглядає клас Time, початково для операції виведення використовується метод Time::display(). Спробувавши отримати результат додавання отримаємо помилку:



Вона виникає, тому що програма не знає яким чином потрібно додавати два об’єкта. Для цього потрібно визначити операцію додавання. Зробимо це всередині класу, додавши один додатковий метод перевантаження операторів. Для цього додамо такий код всередину класу:

```
Time operator +(const Time &t2) {
          return Time(this->_hours + t2._hours,
this->_minutes + t2._minutes);
     }
```

та виведемо на екран результат додавання:



Звідси видно, що результатом додавання стала сума годин та хвилин, яка потім відформатувалась конструктором нового об’єкту t1+t2, який методом Time::display() вивів результат на екран.

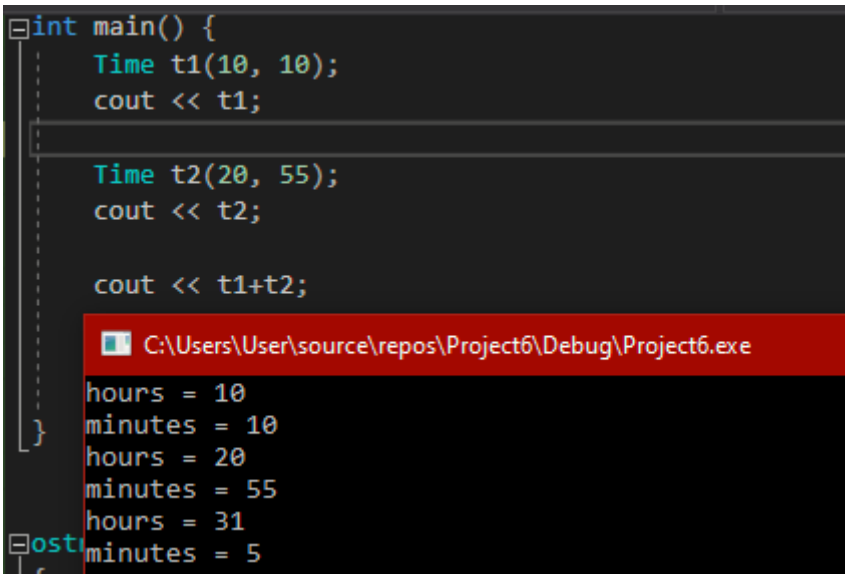
Для зручності виведення, можна перевизначити метод виводу. Для цього потрібно в середині класу дописати таку стрічку:

```
friend ostream & operator <<(ostream &os, const Time &t2);
```

Ключове слово friend дає методу виводу доступ до приватних елементів класу, тому що в нас немає написаних гетерів для цих елементів. Далі за границею класу треба визначити як саме виводити об’єкт, для цього ми додаємо таку стрічку:

```
ostream & operator<<(ostream & os, const Time & t2)
{
     os << "hours = " << t2._hours << endl << "minutes = " << t2._minutes << endl;
     return os;
}
```

Ostream це місце, куди треба вивести цей текст. Тепер наш метод виводу можна замінити на :



Це є ще одна, дуже важлива відмінність в досліджуваних мовах програмування. Також цей метод достатньо простий, щоб його використати замість методу, який буде виконувати ту саму дію або навпаки, що робить його зручним для плагіату.

1.3 Структури

Ще однією відмінністю між Java та C++ є наявність у останньої таких користувацьких типів даних як структура. Це дуже зручний спосіб, коли потрібно швидко створити власний тип даних, в якому буде зберігатись вся важлива інформація та наявність конструкторів, деструкторів, методів та іншого не потрібна.

Для створення структури використовується ключове слово struct, після чого йде її назва, та у душках список всіх потрібних аргументів. Мінусом є те, що кожен раз потрібно окремо ініціалізувати всі аргументи, проте при присвоєнні одній структурі значення іншої всі дані автоматично перекопійовуються. Звичайно структури з різними назвами та наборами аргументів не можна прирівнювати.

В структурі, так само як і в класах, перестановка аргументів не приведе до зміни роботи програми, тому що кожний аргумент потрібно ініціалізувати напряму. Зустрічаються часткові випадки, коли структуру ініціалізують в момент створення, наприклад:

```
struct Student
{
     int age;
     int course;
     int id;
};

int main() {
     Student s = { 17,1,123 };
     cout <<"age = "<< s.age <<" course = "<<s.course<<" id =
"<<s.id<< endl;
     return 0;
    }
```

В цьому випадку зміна порядку аргументів не призведе до помилки в запуску коду, проте відповідь буде розрахована спираючись на іншу послідовність значень. В гіршому випадку, коли буде несумісність типів буде

помилка при компіляції. Ми будемо вважати, що код написаний таким чином, що при однакових аргументах буде виводитись однакова відповідь.

В задачах такої складності, як у попередньому прикладі, де не потрібна велика кількість змінних чи приватність даних не важлива, можна сміливо використовувати структуру замість класу. Цим доволі часто користуються нечесні учні для перетворення чужого коду у перероблений свій. Важливо те, що такий випадок достатньо важко виявити програмними методами. В випадку, коли разом з цим методом використають переіменування змінних та/або заміну типів даних на еквівалентні виявити плагіат майже неможливо як з точки зору перевіряючого, так і з допомогою програмних засобів. Та важко назвати це плагіатом, тому що по суті більша частина перероблена, а без розуміння фундаментальних знань з цієї мови це не вийде.

1.4 Шаблони

Шаблони в C++ є дуже важливою функцією. Шаблони — це загальний опис поведінки функцій, які можна викликати для об'єктів різних типів. Вони потрібні для того, щоб замість того, щоб писати якусь конкретну дію для кожного типу даних записати одну функцію, яка вміє робити ідентичні дії з даними іншого типу.

Записується ця конструкція так:

```
template <typename T> // оголошення параметра шаблона функції типу T
T назва функції(T a) // перелік аргументів
{return a; //повернути результат типу T
}
```

Замість `<typename T>` можна використати `<class T>`. Передавати параметри не обов'язково одному, якщо потрібно передати декілька, потрібно в дужках записати через кому їх назви, ідентично прикладу.

Розглянемо приклад: нам потрібно знайти суму двох елементів одного типу, для трьох різних типів. Якщо б ми не знали про шаблони, потрібно було для кожного типу створювати окремий метод, який вміє працювати з ними. Ми цього робити не будемо, а одразу напишемо код.

```
template<typename T>
const T& add(const T&a, const T&b) {
     return a + b;
}
int main() {
     const int a = 2, b = 3;
     cout <<"a+b = "<< add(a, b) << endl;
     return 0;
    }
```

Запустимо та перевіримо результат

```
template<typename T>
const T& add(const T&a, const T&b) {
    return a + b;
}
int main() {
    const int a = 2, b = 3;
    cout <<"a+b = "<< add(a, b) << endl;
}
```

```
C:\Users\User\source\repos\Project6\Debug\Project6.exe
a+b = 5
Для продолжения нажмите любую клавишу . . .
```

Як бачимо, додавання проведено успішно. Тепер спробуємо додати нестандартний тип даних. Візьмемо з розглянутого прикладу клас Time, та спробуємо знайти суму двох часів.

```
template<typename T>
const T add(T a, const T& b) {
    return a + b;
}
int main() {
    const Time t1(10, 10), t2(20, 55);
    cout << "a+b = " << add(t1, t2) << endl;
}
```

```
C:\Users\User\source\repos\Project6\Debug\Project6.exe
a+b = hours = 31
minutes = 5

Для продолжения нажмите любую клавишу . . .
```

Видно, що ми можемо порахувати суму двох класів. Проте є важливе зауваження: для того, щоб відбулась дія в темплейті, потрібно щоб вона була визначеня для цього типу.

Якщо ми спробуємо до цілочисельного типу додати години, то у нас буде декілька помилок. Перша з них, в нашому темплейті вказаний один тип, тож ми не зможемо замість цілочисельного типу потім використати дробовий, потрібно буде вказати другий тип при написанні темплейту. Інше зауваження, операція додавання або будь-яка інша яку ми захочемо обрати, не визначена для таких пари типів, тож потрібно як і в прикладі перевизначити операцію додавання для даних типів.

Шаблони це дуже корисне розширення. Гарним прикладом використання шаблонів є калькулятор матриць, де потрібно знаходити додавання матриць, множення, ранг та інше. Без шаблонів реалізація буде дуже об’ємна. Дуже легко переписати загальну програму з використанням шаблонів.

1.5 Множинне наслідування

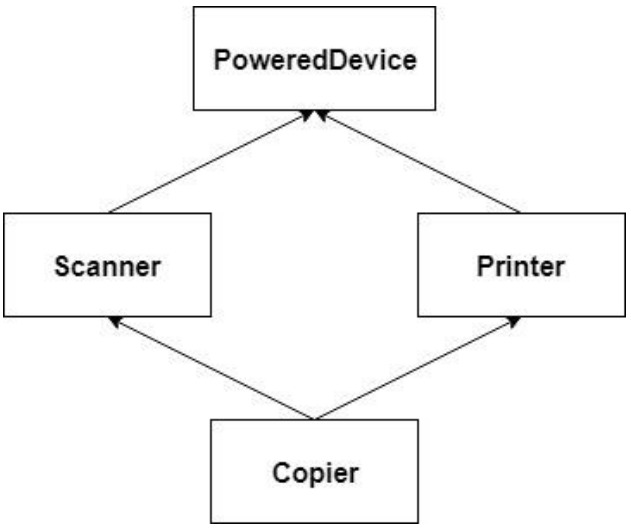
Одиночні успадкування є в багатьох мовах програмування, та C++ підтримує множинне наслідування.

Множинне наслідування чи успадкування дає змогу одному дочірньому класу мати декілька батьків. Широкого поширення ця дія немає, проте все ж вона й досі може бути використана. Це робиться для того, щоб дочірній клас отримав властивості своїх батьківських класів.

Проте в цьому методі є достатньо багато проблем при використанні. Перша проблема – це неоднозначність. Може трапитись ситуація, коли декілька батьківських класів мають наприклад однакову назву методів. При подальшому виклику методу з дочірнього класу, компілятор буде рухатись по ланцюжку зв'язків вгору та шукати відповідний метод. Коли він знайде декілька однакових назв, компілятор покаже помилку. Вирішення цієї проблеми є. Для однозначності потрібно явно вказати яку версію методу потрібно виконати. Проте чим більше батьківських класів, тим більша ймовірність отримати конфлікти імен, які в свою чергу доведеться кожен раз задавати явно.

Більш серйозна проблема виникає, коли один клас має два батьківських класи, кожен з яких має успадкування одного й того ж батьківського класу.

Розглянемо такий випадок на прикладі:



Сканер і принтер обидва отримують живлення з розетки, в той час як ксерокс в своєму розпорядження має властивості і сканера і принтера. Тут виникає проблема не тільки в неоднозначності методів, але й в копіюванні даних з powered в device copier двічі. Цю проблему також можна вирішити за допомогою явного вказування методу, проте програма стає набагато важчою ніж вона може бути.

Є об’єктно-орієнтовані мови програмування, які не підтримують множинне наслідування. В їх число входять і такі сучасні мови програмування як Java та C#, які обмежують кількість наслідувань до одиниці.

В більшості випадків можна обійтись заміною множинного успадкування одиночним, проте є й такі, в яких воно буде кращим. В задачах середнього рівня його використання не потрібне. Проте варто зазначити, що такі об’єкти як std::cin і std::cout бібліотеки iostream, реалізовані з використанням множинного успадкування!

1.6 Оператор переходу goto

Крім операторів переходу, таких як return, break, continue які є в обох мовах, C++ також підтримує оператор goto. Якщо return потрібно використовувати, бо без нього неможливо повернути результат, то інші бажано не використовувати.

Добре продуманий код з циклами та умовами не буде містити їх, тому що в них не буде необхідності. Проте їх використання не заборонене. Оператор goto змушує програму виконати перехід до місця, куди буде вказано.

Розглянемо наступний приклад:

```
int main() {
     double z;
tryAgain: // це місце для переходу
     cout << "Enter a non-negative number: ";
     cin >> z;
     if (z < 0.0)
          goto tryAgain; // а це оператор goto
     cout << "The sqrt of " << z << " is " << sqrt(z) <<endl;
     return 0;

    }
```

Це програма для знаходження кореню з числа у невід'ємній множині. Оператор виконує функцію переходу допоки не буде введено позитивне число чи 0. Таким чином можна переписати умову if за допомогою цього оператору.

Проте краще ним не користуватись. Тому що через використання великої кількості переходів призводить до такого явища, яке називають "Спагеті код". Мається на увазі, що через них виникає розрив коду, читабельність знижується і можна припустити помилку. Проте для перетворення програми без втрати функціональності цей метод підходить.

1.7 Параметри за замовчуванням

Параметри за замовчуванням є ще однією корисною функцією C++. Параметр за замовчуванням – це такий параметр, який не обов'язково ініціалізувати при виклику функції чи конструктору. Відмінністю від звичних нам функцій є знак дорівнює після об'яви кожної змінної і присвоєння їх певного значення у випадку, коли параметр або декілька не були ініціалізовані.

Ця функція корисна, коли через якусь причину вхідним змінним не було присвоєно значення. Потрібно зазначити, що типи параметрів за замовчуванням та типи вхідних змінних повинні бути сумісними.

Є ще один виняток. Параметри за замовчуванням можна взагалі не передавати, проте потрібно щоб вони були в кінці списку, навіть якщо ви будете їх передавати. Тобто, якщо перший аргумент встановити за замовчуванням, то потрібно й встановити всі наступні.

Подивимось наступний приклад. Ми хочемо написати функцію, яка буде підносити число в певну степінь. Якщо вона не задана, то до нульової степені, інакше до тої, яку ми передамо.

Наш метод з параметром за замовчуванням

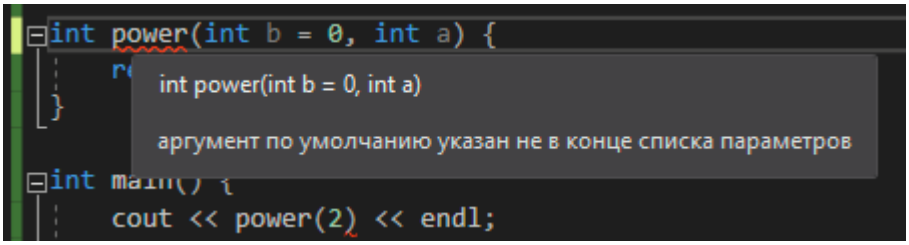
```
int power(int a, int b = 0) {
     return pow(a, b);
    }
```

Тепер спробуємо передати один параметр.

```
int power(int a, int b = 0) {
    return pow(a, b);
}

int main() {
    cout << power(2) << endl;
}
```

```
C:\Users\User\source\repos\Project6\Debug\Project6.exe
1
Для продолжения нажмите любую клавишу . . .
```

Як ми бачимо, передачі другого параметру не було, тож за замовчуванням b = 0. Якщо спробувати переставити місцями аргументи, то отримаємо наступне:



Компілятор не розуміє який саме з параметрів ви передаєте. В цьому випадку зрозуміло, що ми передаємо тільки другий параметр, проте якщо аргументів буде більше, а параметри за замовчуванням будуть перериватись, то при передачі одного параметру за замовчуванням та всіх інших звичайних буде плутанина при передачі.

Розділ 2: що таке лямбда терми та чому вони нам потрібні

Отже лямбда терми або лямбда числення – це формальна система, яка винайдена Алонсо Черчем у тисяча дев'ятсот двадцятому році. В ній всі операції зводяться до елементарних операцій, а саме означенню функції та її визначення.

Дуже корисним в цій мові є те, що лямбда числення можна розглядати як звичайну мову програмування, на якій можна описувати обчислення, так і як математичний об'єкт, на якому можна доводити строгі припущення.

2.1 Основні правила в лямбда термах

В лямбда численні програмний код називається лямбда термом. Для його написання у нас є тільки три основні правила:

1) Змінні x, y, z … є термами
2) Якщо M і N терм, то і (MN) терм
3) Якщо x змінна а M терм, то (λx.M) також терм.

Вводять ще одне правило, що інших термів не існує. У першому правилі пояснюється, шо у нас є алфавіт символів, і що вони є базовими в побудові термів. Друге є правилом застосування функції до аргументу. В ньому M є функцією а N аргумент. Всі функції є функціями одного аргументу, проте вони можуть повертати функції, тож ми можемо робити дії над функціями з декількома аргументами. Якщо ми захотіли б записати синус пі в цьому синтаксисі, то воно виглядало б як (sin PI). Третє правило показує як створювати функції. Конструкція λx.M означає, що ми хочемо створити функцію, яка має аргумент x з тілом функції M.

Правила послідовності дій

Неозброєним оком можна помітити, що будь-який лямбда вираз містить багато дужок. Щоб уникати зайвих використань дужок та знати як їх розкривати введемо наступні правила:

- В послідовності комбінацій(застосувань функцій) приорітет завжди має найближча ліва функція, тобто послідовність E1, E2, …, En можна записати як (… ((E1) E2) … En)
- Область дії заголовку λx поширюється вправо наскільки це можливо, тобто запис λx. E1 E2 … En означає те саме, що й (λx. (E1 E2 … En))
- В послідовності записів лямбда абстракцій приорітет завжди у правого аргументу, тобто λx1…xn. E (λx1. (…xn. E) …). Для прикладу λx y. E означає (λx. (λy. E))

2.2 3 види конверсій для лямбда виразів

Перш ніж вводити поняття цього пункту розглянемо що таке вільна та зв'язана змінна. Входження змінної x в тіло E вільне, коли воно не належить області дії λx, або зв'язна в іншому випадку. Наприклад в лямбда термі (λx. y x) (λy. x y) в першому випадку незалежною змінною є x, у другій дужці – y.

Підстановки

Напишемо наступне твердження. Виклик функції f від аргументу x в лямбда численні виглядає як (f x), отже виклик функції λ.x E від функції G буде виглядати як ((λ.x E) G). Результат такого обчислення буде полягати у підстановці всіх x в виразі E на G і відкидання заголовка функції λ.x. ((λx. E) G) → E [x := G] . Тут вираз [x := G] означає, що ми підставляємо G замість всіх! входжень x у вираз E.

Приклад: . ((λx. xx) z) = xx [x := z] = zz

Тепер про самі підстановки. Змінні x, y … в лямбда численні абстрактні, тож немає різниці між виразами λx. x та λy. y, з таким самим успіхом можна підставити будь-яку іншу змінну. При передачі цим функціям однакових параметрів буде отримане однакове значення.

Розглянемо приклад:

Вирази E1 = λx. xy і E2 = xy. Так як змінні абстрактні, ми можемо у другому виразу замінити змінні. Нехай це будуть u, v. Тепер обчислимо такий вираз: (E1 E2). Якщо підставити першу версію E2 отримаємо (λx. xy xy) = замість x підставляємо xy та прибираємо лямбда = xyy. Тепер застосуємо цей прийом до других пар змінних, отримаємо (λx. xy uv) = підставляємо по аналогії = uvy. Тут вийшла функція від трьох аргументів, попередня вийшла від двох. Причиною цього стало використання однакового набору змінних у двох виразах. Це призвело до конфлікту імен, тому потрібно підбирати змінні, які не будуть призводити до конфлікту імен.

Конверсії

Тепер поговоримо про те, що таке конверсії. Операція конверсії, це така операція, яка призводить до переходу від одного виразу до іншого еквівалентного. Є три типи конверсії.

До першого типу належать альфа конверсії. Лямбда виразу, до якого використовують альфа конверсію називається альфа редексом(від словосполучення REDucible EXpression). Якщо просими словами, то правило альфа конверсії говорить про можливість перейменування зв'язаних змінних, якщо це не призводить до конфлікту імен.

Приклад альфа конверсій, що допускаються:

λx. x → λy. y
λx. f x → λy. f y

Приклад недопустимих альфа конверсій:

λxy. x y не можна конвертувати як λyy. y y .

Другий вид конверсій – бета конверсія. Фактично вона відповідає за обчислення значення функції від аргументу. В якості бета редекса може виступати тільки комбінація.
Приклад допустимих бета конверсій
(λx. f x) E → f E
(λx. x x) (λy. y) → (λy. y) (λy. y) → (λy. y)

Третій вид – ета конверсія. Це правило, за яким можна ввести чи видалити формальний параметр. Редексом для ета конверсій є тільки абстракція. За допомогою цього методу можна ввести наступну властивість: дві функції ідентичні, якщо вони дають однакові результати при однакових вхідних параметрах.

2.3 Рівність лямбда виразів, редукція

Два лямбда вирази рівні, якщо від одного виразу до другого можна перейти за скінченну кількість конверсій. Не потрібно плутати рівність з еквівалентністю, яка означає, що лямбда вирази абсолютно ідентичні. Позначається еквівалентність ≡. Наприклад λx. x = λy. y але λx. x ≢ λy. y. Відношення рівності є симетричним, проте якщо два вирази рівні, то це не означає, що їх можна конвертувати в обидві сторони.

Наприклад, спробуємо конвертувати вираз (λx. x x) f = f f . Вправо вираз можна перетворити бета конверсією, вліво ніяка конверсія не зможе перетворити правий вираз у лівий.

Редукцією лямбда виразів будемо називати перехід від одного лямбда виразу до іншого за скінченну кількість конверсій. Позначається E1 → E2 . Коли до виразу можна застосувати лише альфа конверсію, будемо вважати, що він знаходиться у нормальній формі. Найчастіше редукція призводить до зменшення лямбда терму, проте є й випадки коли розмір збільшується, наприклад:

(λx. x x x) (λx. x x x) → (λx. x x x) (λx. x x x) (λx. x x x) → … .

2.4 Найпоширеніші методи плагіату в мовах C++ та Java

P1: Першим найбільш простим та популярним методом плагіату є перейменування змінних. Цей метод є поширеним не тільки в досліджуваних нами мовами програмування, а й в інших. Припустимо, що у нас є метод, який на вхід отримує дві змінні, які потім потрібно додати та повернути. Приклад такого методу:

```
[O1]:оригінальна програма
int sum(int first_argue, int second_argue){
return first_argue + second_argue;
}.
```

І ми хочемо його змінити. Для цього перейменуємо всі доступні назви, отримаємо, наприклад, наступний метод:

```
[T1]: трансформована програма
int add(int a, int b) { return a + b;}
```

P2: Другий метод – введення додаткової змінної. Ми можемо ввести проміжну змінну яка буде копіювати наші значення або ланцюжок дій буде однаковим як з двома, так і з трьома змінними. В першому розділі ми розглядали створення проміжної змінної з указником чи посиланням, проте тут ми розглянемо більш легкий випадок. Для прикладу візьмемо операцію виведення вхідної змінної та подальша робота з нею. Будемо вважати, що всі змінні були оголошені.

```
[O2]:оригінальна програма
  {
      a = 1;
      cout << a << endl; // подальша робота, її аналіз
зараз не розглядаємо
  }
```

Тепер додамо до цього коду проміжну змінну, отримуємо:

[T2]: трансформована програма

```
{
    a = 1;
    b = a;
    cout << b << endl;
}
```

Так ми можемо отримати той самий результат просто ввівши копію значення та подальша ідентична робота до першого прикладу.

P3: Третій метод – зміна циклу на еквівалентний. Майже жодна програма не може обійтись без циклів. Є декілька різновидів циклів, тож можна переписати код без зміни дієздатності. Наприклад, у нас є програма, яка відпрацьовує певну кількість кроків та повертає кількість виконаних кроків. Запишемо її через цикл for

[O3]:оригінальна програма

```
int iteration() {
    int i = 0;
    for (i = 2; i != 0; i = i - 1) {тіло функції}
    return i;
}
```

Можна цей цикл переписати за допомогою циклу while:

[T3]: трансформована програма

```
int iteration() {
    int i = 2;
    while (i != 0) {
тіло функції
i = i - 1;
}
    return i;
}
```

Відмінністю цих циклів є те, що перший розрахований на відпрацювання конкретну кількість ітерацій, другий більше підходить коли у нас виконується якась умова виходу, проте заміна не буде помилкою.

P4: Четвертий метод – перестановка змінних місцями. Коли зміна порядку виконання не призведе до неправильної роботи програми, ми можемо поміняти відповідні рядки місцями. Для прикладу, візьмемо метод додавання 1 до двох об’єктів:

[O4]:оригінальна програма

```
void plus_1(int first, int second) {

     first = first + 1;
     second = second + 1;
     cout << first << second;
    }
```

Зміна рядків 2 і 3 місцями не призведе до помилки, тож ми можемо зробити перестановку:

[T4]: трансформована програма

```
void plus_1(int first, int second) {

     second = second + 1;
     first = first + 1;
     cout << first << second;
    }
```

Отримуємо той самий результат з іншим кодом.

P5: П’ятий метод – зміна умови з використання операції заперечення. Цей метод найчастіше використовується в if умовах, можна умову замінити на протилежну та додати знак заперечення, або декілька разів викликати операцію заперечення. Розглянемо таку програму:

[O5]:оригінальна програма

```
bool condition(int a) {
     if (true)
          a = 1;
     else
          a = 0;
     return a;
```

```
}
```

Проте ми можемо переписати її з еквівалентною умовою :

[T5]: трансформована програма

```
bool condition(int a) {
     if (!false)
          a = 1;
     else
          a = 0;
     return a;
    }
```

Використання посилань та операції роз’іменування

P6: Приклади для роботи з пам’яттю добре розписані в першій главі. Для подальших досліджень ми візьмемо простіший варіант, щоб показати суть. Отже у нас є програма, яка створює указник, та потім віддає значення об’єкту за цією адресою за допомогою операції роз’іменування. Оригінальна програма має вигляд:

[O6]:оригінальна програма

```
int main() {
     int a = 1;
     cout << "a = " << a << endl;
     return 0;
}
```

[T6]: трансформована програма

```
int main() {
     int a = 1;
     int *b = &a;
     cout << "a = " << *b << endl;
     return 0;
    }
```

2.5 Лямбда-еквівалентність термів

Для розглянутих випадків плагіату P1-P6, нехай терм TOi побудований по оригінальній програмі Oi, а терм TTi побудований по трансформованій програмі Ti, i = 1, .. , 6.

Теорема 1. Для всіх i =1, 2, 3 терми TOi та TTi лямбда еквівалентні

Доведення:

Тож спробуємо довести, що методи плагіату з попереднього розділу можна виявити за допомогою однієї з двох теорем. Проведемо порівняльний аналіз програм.

1Візьмемо перший метод – перейменування змінних та відповідний приклад. У чистому лямбда численні, якого ми дотримуємось, немає чисел, операцій додавання, множення, операторів виведення, циклів та іншого, є лише застосування функцій до аргументів, які також є функціями. Введемо наступне представлення чисел, запропоноване Черчем:

$0 = \lambda\ f\ x.\ x$

$1 = \lambda\ f\ x.\ f\ x$

$n = \lambda\ f\ x.\ f^n\ x$

де f – функція слідування, x – запис 0. Фактично щоб отримати n, потрібно до n-1 числа застосувати функцію слідування, або іншими словами додати одиницю до попереднього результату. Схоже отримується операція додавання та множення:

$n + 1 = \lambda\ n\ f\ x.\ n\ f\ (\ f\ x\ )$

$m + n = \lambda\ m\ n\ f\ x.\ m\ f\ (n\ f\ x)$

$m * n = \lambda\ f\ x.\ m\ (n\ f)\ x$

де m,n – запис чисел у формі Черча, всі введені функції дають результат – число також в цій формі. За основу додавання взяте додавання n раз одиниці до x(нуль), до цього результату потім додати одиницю m разів. За основу

множення взято операція додавання. Запис означає, що ми застосуємо додавання n до нуля та зробимо це m раз.

Тепер ми можемо записати перший вираз в лямбда численні, це буде виглядати як:

O1: λ sum. sum (λ f x. ((λ g y. $g^{\text{first\_argue}}y$) f ((λ h z. $h^{\text{second\_argue}}z$) f x)).

Запишемо другий вираз T1: λ add. add (λ f x. (λ g y. $g^{\text{a}}y$) f ((λ h z. . $h^{\text{b}}z$) b f x)).

Згідно з правилом альфа конверсії, ми можемо перейменувати змінні, наприклад першого виразу, якщо це не призведе до конфлікту імен. Значення на вході функції мають бути однакові, щоб отримати однаковий результат виконання. Ми можемо взяти будь-які назви, для доведення візьмемо ідентичні імена з другого прикладу. Отримаємо наступний терм:

λ sum. sum (λ f x. ((λ g y. $g^{\text{first\_argue}}y$) f ((λ h z. $h^{\text{second\_argue}}z$) f x)). $\xrightarrow{alpha}$
λ add. add (λ f x. (λ g y. $g^{\text{a}}y$) f ((λ h z. . $h^{\text{b}}z$) b f x)

Ми бачимо, що після альфа конверсії ми отримали, що перший вираз еквівалентний другому, а конверсія першого еквівалентна другого. Це означає, що ми довели рівність двох програм, в яких були переіменовані змінні за допомогою лямбда числення.

1Введення додаткової змінної

Припустимо, що нечесний учень зовсім не розуміє що робить програма, та щоб якось її змінити він хоче додати ще одну, або декілька змінних, які змінять код проте не дасть ніякого впливу на результат. За основу візьмемо приклад, який був описаний в відповідному методі плагіату. Перепишемо ці дві програми в лямбда числення:

O2: (λ a. a) (λ f x. f x)

T2: (λ b. b)((λ a. a) (λ f x. f x)) → (λ b. b) (a [a:= λ f x. f x]) $\xrightarrow{beta}$
(λ b. b) (λ f x. f x) $\xrightarrow{alpha}$(λ a. a) (λ f x. f x) .

Ми бачимо, що за допомогою альфа та бета конверсій ми змогли звести лямбда терм переробленої програми до лямбда терму оригіналу, тим самим показавши справедливість теореми 1 для цього випадку.

1Заміна циклу на еквівалентний

Візьмемо приклад з попередньої частини, який відповідає цьому методу. Так як в лямбда численні немає назв функцій, тож як після першої ітерації перейти до наступної не зовсім зрозуміло. Виходом з цього глухого куту є визначення так званої нерухомої точки. Отже терм Y називається комбінатором нерухомої точки, якщо для будь-якого терму f виконується рівність: f (Y f) = Y f.

Для прикладу такого терму, можна привести рекурсивний терм для обрахунку факторіалу:

F = λ f n. if ISZERO n then 1 else n * f(PRE n)
Бачимо, що у випадку, коли на вхід приходить 0, ми повертаємо 1, інакше рахуємо результат цієї функції в наступній точці, тобто в точці n – 1. ISZERO є операцією перевірки числа на рівність 0, Pre – попереднє значення.

Попереднє значення, тобто віднімання одиниці від переданого аргументу функції вводиться наступним чином:

Потрібно задати PRE таке, що PRE 0 = 0 та PRE (n + 1) = n, та PREFN, таку що
PREFN f (true, x) = (false, x) та PREFN f (false, x) = (false, f x)
Записується як PREFN = λ f p. (false, if fst p then snd p else f(snd p)
PRE n = λ f x. snd(n (PREFN f) (true, x))

Так само як задана функція факторіал можна визначити цикл, передавати на вхід число аргументу та на вихід виконувати тіло циклу. Тож запишемо терм T5 спочатку так:

O3 = λ while i. if ISZERO i then 0 else while (Pre i)
Підставимо замість if та умови ISZERO підставимо вирази в лямбда термах. Отримаємо:

$O3 =$ λ while i. (i (λ x. false) true) (0 while(Pre(i)))

Підставимо числа у вираз :

$O3 =$ λ while i. ((λ g y. $g^i y$ (λ x. false) true) (λ g2 y2. $y2$ while(Pre(i)))

Тепер запишемо транспоновану програму. Ми не можемо явно записати цикл for, нам також потрібно зобразити його в вигляді рекурсії. Чим він відрізняється від попереднього циклу ? Відповідь – тільки назвою та формою запису. В ньому оновлення лічильника відбувається після етапу виконання тіла функції. В даному випадку, алгоритм у нас співпадає. Значення вхідних зміних теж однакові, тож ми можемо одразу переписати транспоновану програму, як:

$T3 =$ λ for i. ((λ g y. $g^i y$ (λ x. false) true) (λ g2 y2. $y2$ for (Pre(i)))

$\xrightarrow{alpha}$ λ while i. ((λ g y. $g^i y$ (λ x. false) true) (λ g2 y2. $y2$ while(Pre(i)))

Отримуємо, що терм O5 лямбда еквівалентний T5, що і потрібно було довести.

Об’єднавши всі три попередні результати можна сказати, що теорема 1 доведена для всіх i=1, 2, 3.

Теорема 2. Для всіх i =4, 5, 6 якщо терми TOi еквівалентні TEi та терми TTi також еквівалентні TEi, то терми TOi та TTi лямбда еквівалентні.

Доведення т2: Використовуючи теорему Чорча Россера, для розглянутих випадків плагіату, якщо оригінал та копію програми в лямбда термах можна звести до проміжної, яка буде еквівалентна обом, то оригінал та копія програми будуть лямбда еквівалентні.

2Перестановка рядків місцями

Це є ще одним поширеним видом плагіату. Змінювати рядки місцями можна, якщо це не призведе до порушень в роботі програми. Отже розглянемо ситуацію, коли в нас є два рядки, при заміні яких місцями результат не зміниться. Перепишемо приклад в відповідний лямбда терм:

str 1: $\left(\lambda\ \text{f1 x1.}\left(\lambda\,\text{g1 y1.}\,g1^{first}\ \text{y1}\right)\text{f1}\ (\ (\lambda\,\text{g2 y2.g2 y2}\ )\text{f1 x1})\right)$

str 2: $\left(\lambda\ \text{f2 x2.}\left(\lambda\,\text{h1 k1.}\,h1^{first}\ \text{k1}\right)\text{f2}\ (\ (\lambda\,\text{h2 k2.h2 k2}\ )\text{f2 x2})\right)$

O4: $\lambda\,\text{plus\_1.plus\_1}\ (\lambda\,a\ b.a\ b\ (\ str\ 1\ )(\ str\ 2\ )\ \rightarrow$

$\lambda\,\text{plus\_1.plus\_1}\ (a\ b\ (\ a \coloneqq str\ 1\ )(\ b \coloneqq str\ 2\ ) \xrightarrow{beta}$

$\lambda\,\text{plus\_1.plus\_1}\ (\ str\ 1\ str\ 2\ ) \xrightarrow{beta}$

$\lambda\,\text{plus\_1.plus\_1}\ (\left(\lambda\ \text{f1 x1.}\left(\lambda\,\text{g1 y1.}\,g1^{first}\ \text{y1}\right)\text{f1}\ (\ (\lambda\,\text{g2 y2.g2 y2}\ )\text{f1 x1})\right)$
$\left(\lambda\ \text{f2 x2.}\left(\lambda\,\text{h1 k1.}\,h1^{first}\ \text{k1}\right)\text{f2}\ (\ (\lambda\,\text{h2 k2.h2 k2}\ )\text{f2 x2})\right)\ )$

Тепер запишемо другий терм. В ньому стрічки однакові, лише переставлені, тому str 1 та str 2 залишаються однаковими. Загальний терм по переробленому коду буде виглядати, як:

T4: $\lambda\,\text{plus\_1.plus\_1}\ (a\ b\ (\ a \coloneqq str\ 2\ )(\ b \coloneqq str\ 1\ ) \xrightarrow{beta}$

$\lambda\,\text{plus\_1.plus\_1}(\ str\ 2\ str\ 1\ ) \xrightarrow{beta}$

$\lambda\,\text{plus\_1.plus\_1}\ (\left(\lambda\ \text{f2 x2.}\left(\lambda\,\text{h1 k1.}\,h1^{first}\ \text{k1}\right)\text{f2}\ (\ (\lambda\,\text{h2 k2.h2 k2}\ )\text{f2 x2})\right)$

$\left(\lambda\ \text{f1 x1.}\left(\lambda\,\text{g1 y1.}\,g1^{first}\ \text{y1}\right)\text{f1}\ (\ (\lambda\,\text{g2 y2.g2 y2}\ )\text{f1 x1})\right)\ )$

Якщо подивитись на цей терм, то можна побачити, що стрічка str 1 відносно str 2 і навпаки мають незалежні змінні, тобто при підстановці змінних в перший терм другий не зміниться з точністю до перестановки. Тож ми можемо поміняти місцями стрічки, подальший результат переробленої програми буде виглядати як:

$\lambda\,\text{plus\_1.plus\_1}\ (\left(\lambda\ \text{f1 x1.}\left(\lambda\,\text{g1 y1.}\,g1^{first}\ \text{y1}\right)\text{f1}\ (\ (\lambda\,\text{g2 y2.g2 y2}\ )\text{f1 x1})\right)$
$\left(\lambda\ \text{f2 x2.}\left(\lambda\,\text{h1 k1.}\,h1^{first}\ \text{k1}\right)\text{f2}\ (\ (\lambda\,\text{h2 k2.h2 k2}\ )\text{f2 x2})\right)\ )$

Результатом всіх цих перетворень ми змогли звести терм 1 до проміжного та терм 2, який також еквівалентний проміжному. Тож між собою терми O4, T4 лямбда еквівалентні.

2Зміна умови

Тепер проведемо аналіз програми зі зміною умови з прикладу попереднього розділу. Запишемо її на мові лямбда термів, отримаємо:

O5: λ cond .cond ((λ x y. x) (λ f a. f a) (λ f a. a))

T5: λ cond .cond ((λ x y. y) (λ f a. a) (λ f a. f a)

Тепер спробуємо шляхом перетворень звести обидва терми до іншого, еквівалентного ним.

O5: λ cond .cond ((λ x y. x) (λ f a. f a) (λ f a. a)) $\xrightarrow{beta}$

λ cond .cond ((λ x . x) (λ f a. f a)) $\xrightarrow{beta}$ λ cond .cond (λ f a. f a)

T5: λ cond .cond ((λ x y. y) (λ f a. a) (λ f a. f a) $\xrightarrow{beta}$

λ cond .cond ((λ x . λ f a. f a) (λ f a. a) $\xrightarrow{beta}$ λ cond .cond (λ f a. f a)

Бачимо, що вираз 1 і 2 зведені до проміжного результату, Отже за теоремою Чорча-Россера, програма оригінал та копія є лямбда еквівалентними.

Типізоване лямбда числення

Для того, щоб можна було явно задати тип змінних, потрібно ввести означення типу та правила роботи з ними. Введення типів вирішує проблему парадоксу Рассела.

При введенні типів, кожному терму призначається певний тип, після чого застосування терму s до терму t може бути застосовано лише для сумісних типів. Підтипи та перетворення типів не дозволено! В мовах C++ та Java, якщо в функцію передати не той тип яка вона очікує, проте вхідний тип можна перетворити у той, який чекає функція, то відбувається неявне перетворення типів. Ми таке перетворення типів розглядати не будемо.

Введемо наступне позначення t:s буде позначати, що t належить тип s. Далі введемо таке позначення f:s→a, яке означає, що функція f відображає множину s в множину a. Проте після введення типів ми будемо розглядати систему як формальну.

Є два основних підходи для використання типів. Перший – це явне вказування типів запропоноване Черчем. Друге, яке ми будемо використовувати – неявна типізація, запропонована Каррі. В цьому випадку структура термів буде розглядатись як нетипізований випадок, проте сам терм може як не мати жодного типу, так і мати його, і не один. Наприклад, функції отримання константи може бути приписаний будь-який тип. Теорема про збереження типу, якої ми будемо дотримуватися, показує, що тип в ході перетворень не змінюється.

Використання посилань та операції роз'іменування

Для цього розділу потрібно сказати, що для типізованого випадку всі властивості перетворень залишаються, як і залишається справедливою теорема Чорча—Россера. Будь-який типізований терм має нормальну форму, а будь-яка можлива послідовність редукцій над типізованим лямбда термом обов'язково скінченна.

Як вже було розглянуто в розділі 1, програму, написану на мові C++ можна легко транспонувати, використавши роботу з пам'яттю. Для цього потрібно ввести операцію взяття посилання, будемо позначати її ref x, та операцію роз'іменування, позначимо її !. Зрозуміло, що операція роз'іменування що застосована до посилання буде просто видавати об'єкт, який зберігається під тим посиланням. Що в оригінальній програмі, що в транспонованій програмі, ні перетворень типів ні конфлікту типів немає.

Напишемо якій програмі відповідають які змінні та їх тип.

Програма	Змінна 1	Змінна 2
оригінал	a:int	-
транспонована	a:int	b:int

Переведемо приклад Р6 у лямбда числення, отримаємо:

О6: $(\lambda f\, a.f\ a) \xrightarrow{alpha} (\lambda f\, y.f\ y)$

Т6: $\lambda !\ ref.(\lambda g.!g(\lambda c.\ ref\, c(\lambda b.b\ (\lambda f\, a.f\ a)))) \xrightarrow{beta}$

$\lambda !\ ref.(!\lambda c.\ ref\, c(\lambda b.b\ (\lambda f\, a.f\ a))) \xrightarrow{beta}$

$\lambda !\ ref.(!ref\ (\lambda b.b\ (\lambda f\, a.f\ a)) \longrightarrow (\lambda b.b\ (\lambda f\, a.f\ a))) \xrightarrow{beta}$

$(\lambda f\, a.f\ a) \xrightarrow{alpha} (\lambda f\, y.f\ y)$

Отже, шляхом перетворень ми звели вираз оригінал до проміжного та вираз транспонованого оригіналу до такого самого проміжного, тож оригінал і транспонована копія є лямбда еквівалентні. Тож отримавши попередні результати можна сказати, що теорема 2 доведена для всіх i=4, 5, 6.

Висновок

Метою цієї роботи було виявлення потенційно схожих програм, написаних на мові C++ та Java. В першому розділі було розібрано тему схожості цих двох мов. Розглянуто сім найважливіших різниць в написанні коду та проілюстровано на прикладах.

В другому розділі було викладено пояснення щодо того, що таке лямбда числення. Також було показано яким чином можна представити код у лямбда численні, введені правила яким чином можна перейти до еквівалентного виразу. Також ми навчились цими методами доводити, що програма оригінал, та транспонована програма за принципом викладених методів плагіату є еквівалентні. На основі цих правил ми сформулювали дві теореми, які потім довели.

В реальності, не можна з абсолютною точністю сказати чи програма була списана, чи просто вона настільки легка, що інших способів зробити її немає, або ж вони просто складні, і їх ніхто не використовує.

Нашою задачею було перевести програму у лямбда числення і досліджувати семантику двох схожих програм через лямбда – еквівалентність відповідних термів. Все що було заплановано на даному етапі роботи було виконано.

До дипломної роботи також додається додаток до дипломної роботи, в якому написані основні операції з програмування та метод їх представлення за допомогою лямбда числення.

Список використаних джерел

Використані джерела до розділу 1:

Відмінність мов https://blog.ithillel.ua/articles/raznica-mezhdu-yazykami-programmirovaniya-c-i-java

Глибше пояснення деяких відмінностей мов

https://acode.com.ua/urok-169-mnozhynne-spadkuvannya/

https://ravesli.com/urok-133-peregruzka-operatorov-vvoda-i-vyvoda/#toc-0

https://ravesli.com/urok-134-peregruzka-operatorov-cherez-metody-klassa/

https://ravesli.com/urok-173-shablony-funktsij/

https://metanit.com/cpp/tutorial/5.14.php

https://ravesli.com/urok-66-operator-goto/

Розділ 2:

Теорія лямбда числення

https://nsu.ru/xmlui/bitstream/handle/nsu/8874/Harrison.pdf;jsessionid=EDE44697994F02AFE84342EA5A6163AA?sequence=1

http://e-maxx.ru/sgu/files/orel.pdf

https://anton-k.github.io/ru-haskell-book/book/14.html

https://habr.com/ru/post/215807/

https://alexeykalina.github.io/technologies/lambda.html

http://www.lib.uniyar.ac.ru/edocs/iuni/20180409.pdf

https://thesz.livejournal.com/11693.html

https://fb.ru/article/462633/lyambda-ischislenie-opisanie-teoremyi-osobennosti-primeryi

https://newstar.rinet.ru/~goga/tapl/tapl007.html